

# DroidGen: Constraint-based and Data-Driven Policy Generation for Android

Mohamed Nassim Seghir<sup>1</sup> and David Aspinall<sup>2</sup>

<sup>1</sup> University College London

<sup>2</sup> University of Edinburgh

**Abstract.** We present DroidGen a tool for automatic anti-malware policy inference. DroidGen is data-driven: uses a training set of malware and benign applications and makes call to a constraint solver to generate a policy under which a maximum of malware is excluded and a maximum of benign applications is allowed. Preliminary results are encouraging. We are able to automatically generate a policy which filters out 91% of the tested Android malware. Moreover, compared to black-box machine learning classifiers, our method has the advantage of generating policies in a declarative readable format. We illustrate our approach, describe its implementation and report on experimental results.

## 1 Introduction

Security on Android is enforced via permissions giving access to resources on the device. These permissions are often too coarse and their attribution is based on an all-or-nothing decision in the vast majority of Android versions in actual use. Additional security policies can be prescribed to impose a finer-grained control over resources. However, some key questions must be addressed: who writes the policies? What is the rationale behind them? An answer could be that policies are written by experts based on intuition and prior knowledge. What can we do then in the absence of expertise? Moreover, are we sure that they provide enough coverage?

We present DroidGen a tool for the systematic generation of anti-malware policies. DroidGen is fully automatic and data-driven: it takes as input two training sets of benign and malware applications and returns a policy as output. The resulting policy represents an optimal solution for a constraint satisfaction problem expressing that the discarded malware should be maximized while the number of excluded benign applications must be minimized. The intuition behind this is that the solution will capture the maximum of features which are specific to malware and less common to benign applications. Our goal is to make the generated policy as general as possible to the point of allowing us to make decisions regarding new applications which are not part of the training set.

In addition to being fully push-button, DroidGen is able to generate a policy that filters out 91% of malware from a representative testing set of Android applications with only a false positive rate of 6%. Moreover, having the policies

in a declarative readable format can boost the effort of the malware analyst by providing diagnosis and pointing her to suspicious parts of the application. In what follows we present the main ingredients of DroidGen, describe their functionality and report on experimental results.

## 2 Application Abstraction

DroidGen proceeds in several phases: application abstraction, constraint extraction and constraint solving, see Figure 1. Our goal is to infer policies that distin-

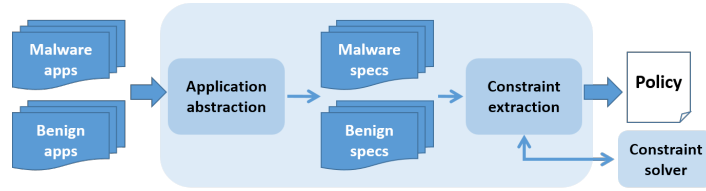


Fig. 1: Illustration of DroidGen’s Main Ingredients

guish between good and bad behaviour. As it is not practical to have one policy per malicious application, we need to identify common behaviours of applications. Hence the first phase of our approach is the derivation of specifications (abstractions) which are general representations of applications. Given an application  $A$ , the corresponding high level specification  $\text{Spec}(A)$  consists of a set of properties  $\{p_1, \dots, p_k\}$  such that each property  $p$  has the following grammar:

$$\begin{aligned}
 p &:= c : r \\
 c &:= \text{entry\_point} \mid \text{activity} \mid \text{service} \mid \text{receiver} \\
 &\quad \mid \text{onclick\_handler} \mid \text{ontouch\_handler} \mid lc \\
 lc &:= \text{oncreate} \mid \text{onstart} \mid \text{onresume} \mid \dots \\
 u &:= \text{perm} \mid \text{api}
 \end{aligned}$$

A property  $p$  describes a context part  $c$  in which a resource  $r$  is used. The resource part can be either a permission  $\text{perm}$  or  $\text{api}$  referring to an api method identifier which consists of the method name, its signature and the class it belongs to. The context  $c$  can be  $\text{entry\_point}$  referring to all entry points of the app,  $\text{activity}$  representing methods belonging to activities,  $\text{service}$  for methods belonging to service components<sup>3</sup>, etc. We also have  $\text{onclick\_handler}$  and  $\text{ontouch\_handler}$  respectively referring to click and touch event handlers. Moreover,  $c$  can be an activity life-cycle callback such as  $\text{oncreate}$ ,  $\text{onstart}$ , etc.<sup>4</sup> Activity callbacks as well as the touch and click event handlers are also entry points.

A property  $p$  of the form  $c : r$  belongs to the specification of an application  $A$  if  $r$  ( $\text{perm}$  or  $\text{api}$ ) is used within the context  $c$  in  $A$ . In other words: it exists at

<sup>3</sup>  $\text{activity}$ ,  $\text{service}$  and  $\text{receiver}$  are some of the building blocks of Android applications.

<sup>4</sup> Some components have a life-cycle governing their callbacks invocation.

least one method matching  $c$  from which  $r$  is transitively called (reachable). To address such a query, we compute the transitive closure of the call graph [9]. We propagate permissions (APIs) backwards from callees to callers until we reach a fixpoint.

For illustration, let us consider the example in Figure 2. On the left hand side, we have code snippets representing a simple audio recording application named `Recorder` which inherits from an `Activity` component. On the right hand side, we have the corresponding specifications in terms of APIs (Figure 2(a)) and in terms of permissions (Figure 2(b)). The method `setAudioSource`, which sets the recording medium for the media recorder, is reachable (called) from the `Activity` life-cycle method `onCreate`, hence we have the entry `oncreate: setAudioSource` in the specification map (a). We also have the entry `oncreate: RECORD_AUDIO` in the permission-based specification map (b) as the permission `RECORD_AUDIO` is associated with the API method `setAudioSource` according to the Android framework implementation. Similarly, the API method `setOutputFile` is associated with the context `onclick` (a) as it is transitively reachable (through `startRecording`) from the click handler method `onClick`. Hence permission `WRITE_EXTERNAL_STORAGE`, for writing the recording file on disk, is also associated with `onclick` (b). Both APIs and permissions are also associated with the context `activity` as they are reachable from methods which are activity members. We use results from [3] to associate APIs with the corresponding permissions.

<pre>public class Recorder extends Activity     implements OnClickListener{     private MediaRecorder myRecorder;     ...     public void onCreate(...) {         myRecorder = new MediaRecorder();         // uses RECORD_AUDIO permission         myRecorder.setAudioSource (...);     }      private void startRecording() {         // uses WRITE_EXTERNAL_STORAGE         myRecorder.setOutputFile (...);         recorder.start ();     }      public void onClick (...) {         startRecording ();     } }</pre>	<table><tr><th>Spec(Recorder)</th></tr><tr><td>oncreate: setAudioSource</td></tr><tr><td>onclick: setOutputFile</td></tr><tr><td>activity: setAudioSource</td></tr><tr><td>activity: setOutputFile</td></tr></table> <p>(a)</p>	Spec(Recorder)	oncreate: setAudioSource	onclick: setOutputFile	activity: setAudioSource	activity: setOutputFile
Spec(Recorder)						
oncreate: setAudioSource						
onclick: setOutputFile						
activity: setAudioSource						
activity: setOutputFile						
	<table><tr><th>Spec(Recorder)</th></tr><tr><td>oncreate: RECORD_AUDIO</td></tr><tr><td>onclick: WRITE_EXTERNAL_STORAGE</td></tr><tr><td>activity: RECORD_AUDIO</td></tr><tr><td>activity: WRITE_EXTERNAL_STORAGE</td></tr></table> <p>(b)</p>	Spec(Recorder)	oncreate: RECORD_AUDIO	onclick: WRITE_EXTERNAL_STORAGE	activity: RECORD_AUDIO	activity: WRITE_EXTERNAL_STORAGE
Spec(Recorder)						
oncreate: RECORD_AUDIO						
onclick: WRITE_EXTERNAL_STORAGE						
activity: RECORD_AUDIO						
activity: WRITE_EXTERNAL_STORAGE						

Fig. 2: Code snippets sketching a simple audio recording application together with the corresponding specifications based on APIs (a) and based on permissions (b)

### 3 Specifications to Policies: an Optimisation Problem?

DroidGen tries to derive a set of rules (policy) under which a maximum number of benign applications is allowed and a maximum of malware is excluded. This is an optimization problem with two conflicting objectives. Consider

$$\begin{array}{l|l} \text{Spec}(\text{benign}_1) = \{p_a\} & \text{Spec}(\text{malware}_1) = \{p_a, p_b\} \\ \text{Spec}(\text{benign}_2) = \{p_c\} & \text{Spec}(\text{malware}_2) = \{p_a, p_c\} \\ \text{Spec}(\text{benign}_3) = \{p_b, p_e\} & \text{Spec}(\text{malware}_3) = \{p_d\} \end{array}$$

Each application (benign or malware) is described by its specification consisting of a set of properties ( $p_i$ 's). As seen previously, a property  $p_i$  can be for example `activity : record_audio`, meaning that the permission `record_audio` is used within an activity. A policy excludes an application if it contradicts one of its properties. We want to find the policy that allows the maximum of benign applications and excludes the maximum of malware. This is formulated as:

$$\text{Max}[\underbrace{I(p_a) + I(p_c) + I(p_b \wedge p_e)}_{\text{benign}} - \underbrace{(I(p_a \wedge p_b) + I(p_a \wedge p_c) + I(p_d))}_{\text{malware}}]$$

where  $I(x)$  is the function that returns 1 if  $x$  is true or 0 otherwise. This type of optimization problems where we have a mixture of theories of arithmetic and logic can be efficiently solved via an SMT solver such as Z3 [7]. It gives us the solution:  $p_a = 0$ ,  $p_b = 1$ ,  $p_c = 1$ ,  $p_d = 0$  and  $p_e = 1$ . Hence, the policy will contain the two rules  $\neg p_a$  and  $\neg p_d$  which filter out all malware but also exclude the benign application *benign*<sub>1</sub>. A policy is violated if one of its rules is violated.

**Policy Verification and Diagnosis.** Once we have inferred a policy, we want to use it to filter out applications violating it. A policy  $P = \{\neg p_1, \dots, \neg p_k\}$  is violated by an application  $A$  if  $\{p_1, \dots, p_k\} \cap \text{Spec}(A) \neq \emptyset$ , meaning that  $A$  contradicts (violates) at least one of the rules of  $P$ . In case of policy violation, the violated rule, e.g.  $\neg p$ , can give some indication about a potential malicious behaviour. DroidGen maps back the violated rule to the code in order to have a view of the violation origin. For  $p = (c : u)$ , a sequence of method invocations  $m_1, \dots, m_k$  is generated, such that  $m_1$  matches the context  $c$  and  $m_k$  invokes  $u$ .

### 4 Implementation and Experiments

DroidGen<sup>5</sup> is written in Python and uses Androguard<sup>6</sup> as front-end for parsing and decompiling Android applications. DroidGen automatically builds abstractions for the applications which are directly accepted in APK binary format. This process takes around 6 seconds per application. An optimization problem in terms of constraints over the computed abstractions is then generated and

<sup>5</sup> [www0.cs.ucl.ac.uk/staff/n.seghir/tools/DroidGen](http://www0.cs.ucl.ac.uk/staff/n.seghir/tools/DroidGen)

<sup>6</sup> <https://github.com/androguard>

the Z3 SMT solver is called to solve it. Finally, the output of Z3 is interpreted and translated to a readable format (policy). Policy generation takes about 7 seconds and its verification takes no more than 6 seconds per app on average.

We derived two kinds of policies based on a training set of 1000 malware applications from Drebin<sup>7</sup> and 1000 benign ones obtained from Intel Security (McAfee). The first policy  $P_p$  is solely based on permissions and is composed of 65 rules. The other policy  $P_a$  is exclusively based on APIs and contains 152 rules. Snippets from both policies are illustrated in the appendix. We have applied the two policies to a testing set of 1000 malware applications and 1000 benign ones (different from the training sets) from the same providers. Results are summarised in Table 1. The policy  $P_a$  composed of rules over APIs performs

Policy	Malware filtered out	Benign excluded
APIs ( $P_a$ )	910/1000	59/1000
Permission ( $P_p$ )	758/1000	179/1000

Table 1: Results for a permissions-based policy ( $P_p$ ) vs. an API-based one ( $P_a$ )

better than the one that uses permissions in terms of malware detection as it is able to filter out 91% of malware while  $P_p$  is only able to detect 76%. It also has a better false positive rate as it only excludes 6% of benign applications, while  $P_p$  excludes 18%. Being able to detect 91% of malware is encouraging as it is comparable to the results obtained with some of the professional security tools (<https://www.av-test.org/>)<sup>8</sup>. Moreover, our approach is fully automatic and the actual implementation does not exploit the full expressiveness of the policy space as we only generate policies in a simple conjunctive form. We plan to further investigate the generation of policies in arbitrary propositional forms.

## 5 Related Work

Many tools for analysing various security aspects of Android have emerged [2, 5, 6, 8]. They either check or enforce certain security properties (policies). These policies are either hard-coded or manually provided. Our work complements such tools by providing the automatic means for inferring the properties to be checked. Hence, DroidGen can serve as a front-end for a verification tool such as EviCheck [9] to keep the user completely out of the loop.

Also various machine-learning-based approaches have been proposed for malware detection [1, 4, 10, 11]. While some of them outperform our method, We did not yet exploit the entire power of the policy language. We are planning to allow more general forms for the rules, which could significantly improve the results. Moreover, many of the machine-learning based approaches do not provide further indications about potential malicious behaviours in an application. Our

<sup>7</sup> <http://user.informatik.uni-goettingen.de/~darp/drebin/>

<sup>8</sup> We refer to AV-TEST benchmarks dated September 2014 as our dataset was collected during the same period.

approach returns policies in a declarative readable format which is helpful in terms of code inspection and diagnosis. Some qualitative results are reported in the appendix. To the best of our knowledge, our approach is unique for being data-driven and using a constraint solver for inferring anti-malware policies.

## 6 Conclusion and Future Work

We have presented DroidGen a tool for the automatic generation of anti-malware policies. It is data-driven and uses a constraint solver for policy inference. DroidGen is able to automatically infer an anti-malware policy which filters out 91% of the tested malware with the additional benefit of being fully automatic. Having the policies in declarative readable format can boost the effort of the malware analyst by pointing her to suspicious parts of the application. As future work, we plan to generate more expressive policies by not restricting their form to conjunctions of rules. We also plan to generate anti-malware policies for malware families with the goal of obtaining semantics-based signatures (see appendix).

## References

1. D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, and K. Rieck. DREBIN: effective and explainable detection of android malware in your pocket. In *NDSS*, 2014.
2. S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. L. Traon, D. Octeau, and P. McDaniel. Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *PLDI*, page 29, 2014.
3. K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie. Pscout: analyzing the Android permission specification. In *CCS*, pages 217–228, 2012.
4. V. Avdiienko, K. Kuznetsov, A. Gorla, A. Zeller, S. Arzt, S. Rasthofer, and E. Bodden. Mining Apps for Abnormal Usage of Sensitive Data. In *ICSE*, pages 426–436, 2015.
5. M. Backes, S. Gerling, C. Hammer, M. Maffei, and P. von Styp-Rekowsky. Appguard - enforcing user requirements on Android apps. In *TACAS*, pages 543–548, 2013.
6. E. Chin, A. P. Felt, K. Greenwood, and D. Wagner. Analyzing inter-application communication in Android. In *MobiSys*, pages 239–252, 2011.
7. L. M. de Moura and N. Bjørner. Z3: An efficient smt solver. In *TACAS*, pages 337–340, 2008.
8. S. Fahl, M. Harbach, T. Muders, M. Smith, L. Baumgärtner, and B. Freisleben. Why Eve and Mallory love Android: an analysis of Android SSL (in)security. In *ACM Conference on Computer and Communications Security*, pages 50–61, 2012.
9. M. N. Seghir and D. Aspinall. Evicheck: Digital evidence for android. In *ATVA*, pages 221–227, 2015.
10. C. Yang, Z. Xu, G. Gu, V. Yegneswaran, and P. A. Porras. Droidminer: Automated mining and characterization of fine-grained malicious behaviors in android applications. In *ESORICS*, pages 163–182, 2014.
11. W. Yang, X. Xiao, B. Andow, S. Li, T. Xie, and W. Enck. Appcontext: Differentiating malicious and benign mobile app behaviors using context. In *ICSE*, pages 303–313, 2015.